intel®

APPLICATION NOTE

AP-143

March 1982

Using the iAPX 86/20 Numeric Data Processor in a Small Business Computer

Ken Shoemaker
Microcomputer Applications

Order Number: 210383-001

INTRODUCTION

As the performance of microcomputers has improved, the types of functions performed by these microcomputers have grown. One application filled by these machines has been to perform typical "adding machine" type calculations, balancing ledgers, etc. This type of machine has come to be called a "small business computer." To be a true business computer, however, the types of operations performed by these machines need to be expanded beyond simple "balance the books" types of operations. There are many algorithms that have been impractical for these small business computers because the number of calculations required by the algorithms and the performance available from these machines did not make them feasible. Such operations were available only on large mainframe or minicomputers. With the introduction of the iAPX 86/20, a microcomputer can finally perform these types of calculations at a cost level appropriate to small business computers.

The iAPX 86/20 features the Intel 8086 with the 8087 numerics co-processor. This combination allows for high-performance, high-precision numeric calculations. Many types of operations require this performance to provide accurate results in a reasonable amount of time. This increased performance will also be particularly welcome in the interactive user environment typically found in small business computers. It is very frustrating to wait many seconds or even minutes after hitting "return" for the computer to generate results.

In general, if there are many methods to solving a business computer problem, the method requiring the largest number or calculations will provide the best results. In many applications, approximate methods have been used because the speed of the hardware (or the cost of the computer time) did not allow a more exact method to be used. Because of the high performance of the iAPX 86/20, these numeric intensive methods may now be used in small business computer software.

The types of calculations demonstrated in this note are:

- Interest and Annuities. These calculations require the use of floating point multiplication, division, exponentiation and logarithms. These calculations are used to determine the present or future value of certain types of funds.
- Restocking. These iterative calculations require extensive use of floating point multiplication and division. They are used to determine the optimum restocking times for a given item when the set-up charges, holding costs and demand for the item are known or can be estimated.
- Linear Programming. These calculations require extensive use of floating point multiplication and division. One of many applications for linear programming is the determination of optimum production quantities of diverse products when the quantities of their various constituents are both overlapping and limited.

iAPX 86/20 HARDWARE OVERVIEW

The iAPX 86/20 is a 16-bit microprocessor based on the Intel 8086 CPU. The 8086 CPU features eight internal general-purpose 16-bit registers, memory segmentation, and many other features allowing for efficient code generation from high-level language compilers. When augmented with the 8087, it becomes a vehicle for high-speed numerics processing. The 8087 adds eight 80-bit internal floating point registers, and a floating point arithmetic logic unit (ALU) which can speed floating point operations up to 100 times over other software floating point simulators or emulators.

The 8086 and 8087 execute a single instruction stream. The 8087 monitors this stream for numeric instructions. When a numeric instruction is decoded, the 8086 generates any needed memory addresses for the 8087. The 8087 then begins instruction execution automatically. No other software interface is required, unlike other floating point processors currently available where, for example, the main processor must explicitly write the floating point numbers and commands into the floating point unit. The 8086 then continues to execute non-numeric instructions until another 8087 instruction is encountered, whereupon it must wait for the 8087 to complete the previous numeric instruction. The overlapped 8086 and 8087 processing is known as concurrency. Under ideal conditions, it effectively doubles the throughput of the processor. However, even when a steady stream of numeric instructions is being executed (meaning there is no concurrency), the numeric performance of the 8087 ALU is much greater than that of the 8086 alone.

The hardware interface between the 8086 and the 8087 is equally simple. Hardware handshaking is performed through two sets of pins. The RQ/GT pin is used when the 8087 needs to transfer operands, status, or control information to or from memory. Because the 8087 can transfer information to and from memory independent of the 8086, it must be able to become the "bus master," that is, the processor with read and write control of all the address, data and status lines. Only one unit is permitted to have control of these lines at a time; chaos would exist otherwise, like four people talking at once with each trying to understand the others.

The TEST/BUSY pin is used to manage the concurrency mentioned above. Whenever the 8087 is executing an instruction, it sets the BUSY pin on high. A single 8086 instruction (the WAIT instruction) tests the state of this pin. If this pin is high, the WAIT instruction will cause the 8086 to wait until the pin is returned to low. Therefore, to insure that the 8086 does not attempt to fetch a numeric instruction while the 8087 is still working on a previous numeric instruction, the WAIT instruction needs to be executed. The 8086/87/88 assembler, in addition to all Intel compilers, automatically inserts this WAIT instruction before most numeric instructions. Software polling can be used to determine the state of the BUSY pin if hardware handshaking is not desired.

Most other lines (address, status, etc.) are connected directly in parallel between the 8086 and the 8087. An exception to this is the 8087 interrupt pin which must be routed to an external interrupt controller. An example iAPX 86/20 system is shown in Figure 1. A more complete discussion of both the handshaking protocol between the 8086 and the 8087 and the internal operation of the 8087 can be found in the application note *Getting Started With the Numeric Data Processor,* AP-113 by Bill Rash, or by consulting the numerics section of the July 1981 *iAPX 86,88 Users Manual.*

In addition to the 8087 hardware, the 8086 is also supported by Intel compilers for both Pascal and FORTRAN. Code generated by these compilers can easily be combined with code generated from the other compiler, from the Intel 8086/87/88 macro assembler or the Intel PL/M compiler. In addition, these compilers produce in line code for the 8087 when numeric operations are required. By producing in line code rather than calls to floating point routines, the software overhead of an unnecessary procedure call and return is eliminated. The combination of both hardware co-processors and software support for the iAPX 86/20 provides for greater performance of both the end product, and its development effort.

ROUTINES IMPLEMENTED

All routines implemented in this application note were written entirely in either Pascal 86 or FORTRAN 86. In addition, a FORTRAN program available from IMSL[1] for use in solving linear programs was used. In each

[1]IMSL, Inc., Sixth Floor-NBC Building, 7500 Bellaire Boulevard, Houston, Texas, 77036. (713) 722-1927.

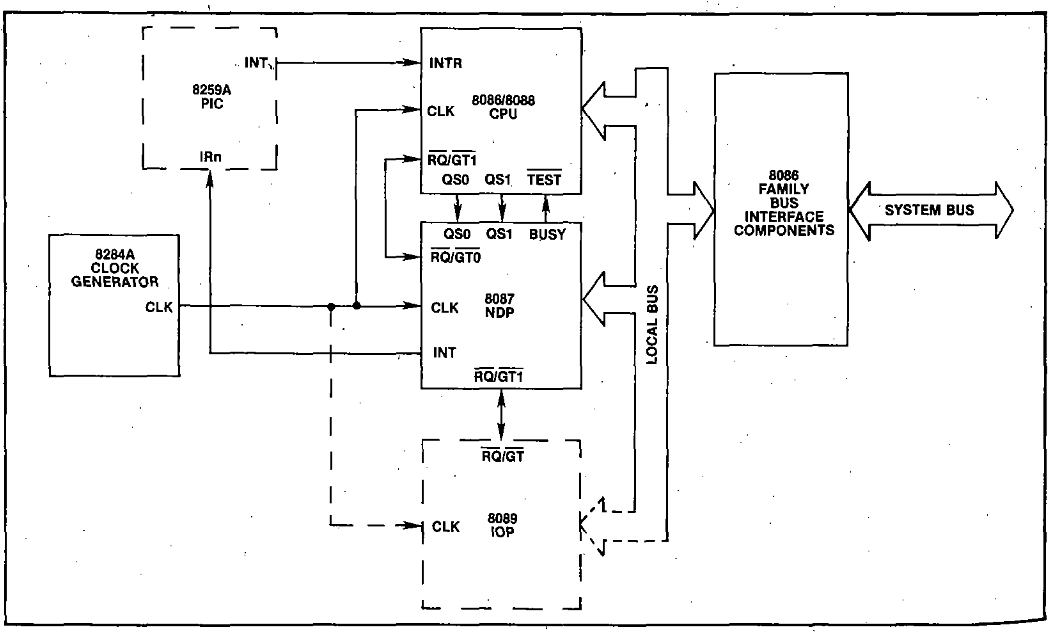

Figure 1. Typical 86/20 System

case, the routine was executed using a 5 MHz iAPX 86/20 on an iSBC86/12 board contained within an Intel Intellec™ Series III development system. The programs can be executed on any iAPX 86/20 (or iAPX 88/20) with sufficient memory, however. In general, the memory requirements for the programs were not substantial. Source listings for all routines written for this note are located in the appendix.

All routines were run using both the 8087 and the 8087 software emulator. The 8087 software emulator is a software package exactly emulating the internal operation of the 8087 using 8086 instructions. When the emulator is used, an 8087 is not required. The emulator is a software product available from Intel as part of the 8087 support library. The performance of the 8087 hardware is much better than that of the software emulator, as one would expect from a specialized floating point unit.

In some routines, values are quoted for the various data formats supported by the 8087. For real numbers, these formats are short real, long real, and temporary real. The differences among the three are in the number of bits allocated to represent a given floating point number.

In all real numbers, the data is split into three fields: the sign bit, the exponent field and the mantissa field. The sign bit indicates whether the number is positive or negative. The exponent and mantissa together provide the value of the number: the exponent providing the power of two of the number, and the mantissa providing the "normalized" value of the number. A "normalized" number is one which always lies within a certain range. By dividing a number by a certain power of two, most numbers can be made to lie between the numbers 1 and 2. The power of two by which the number must be divided to fit within this range is the exponent of the number, and the result of this division is the mantissa. This type of operation will not work on all numbers (for example, no matter what one divides zero by, the result is always zero), so the number system must allow for these certain "special cases."

As the size of the exponent grows, the range of numbers representable also grows, that is, larger and smaller numbers may be represented. As the size of the mantissa grows, the resolution of the points within this range grows. This means the distance between any two adjacent numbers decreases, or, to put it another way, finer detail may be represented. Short real numbers provide eight exponent bits and 23 significand or mantissa bits. Long real numbers provide 11 exponent bits and 52 significand bits. Temporary real numbers provide 15 exponent bits and 63 significand bits. These data formats are shown in Figure 2. Thus, of the three data formats implemented, short real provides the least amount of precision, while temporary real provides the greatest amount of precision. These levels of precision represent only the external mode of storage for the numbers; inside the 8087 all numbers are represented in temporary real precision. Numbers are automatically converted into the temporary real precision when they are loaded into the 8087. In addition to real format numbers, the 8087 automatically converts to and from external variables stored as 16, 32 or 64-bit integers, or 80-bit binary coded decimal (BCD) numbers.

Memory requirements also increase as precision increases. Whereas a short real number requires only four bytes of storage (32 bits), a long real number requires eight bytes (64 bits) and a temporary real number 10

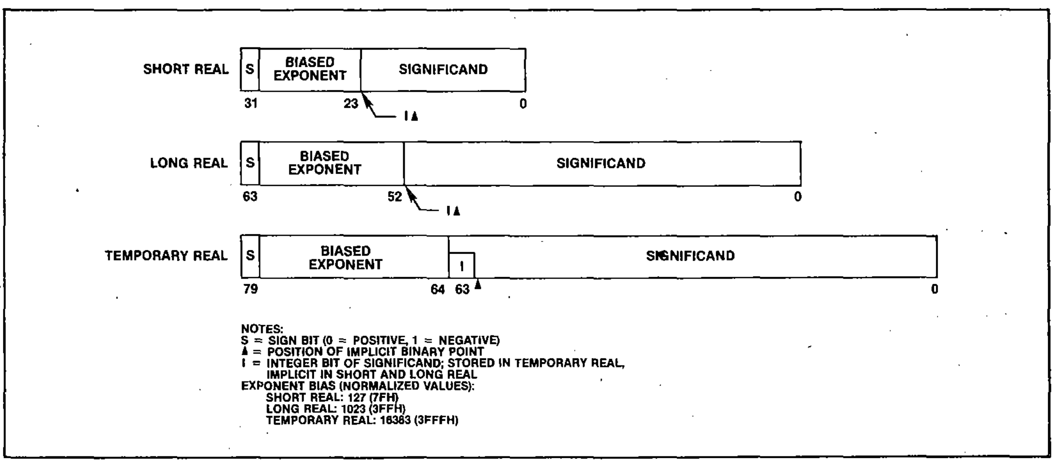

Figure 2. Data Formats

AFN-02184A

bytes (80 bits)! In many floating point processors, processing time also increases dramatically as precision is increased, making this another consideration in the choice of precision to be used by a routine. The differences in 8087 processing time among short real, long real and temporary real numbers is relatively insignificant, however. This makes the choice of which precision to use in an iAPX 86/20 system a function only of memory limitations and precision requirements.

Interest

Routines were written to calculate the final value of a fund when given the annual interest and the present value. Although the calculations required to generate individual interest values are rather short, the additional precision of the iAPX 86/20 can be used to generate better results. In addition, if a large number of interest calculations are to be performed (or if an interest rate type of calculation is used as part of an iterative model), the speed of the single interest rate calculation is important, as it will be performed very many times.

It is assumed that the interest will be compounded daily, which requires the calculation of the yearly effective rate. This value, which is the equivalent annual interest rate when interest is compounded daily, is determined by the following formula:

$$yer = (1 + (\frac{i}{np}))^{np} - 1$$

Where:

- **yer** is the yearly effective rate
- **i** is the annual interest rate
- **np** is the number of compounding periods per annum

Once the **yer** is determined, the final value of the fund can be determined by:

$$fv = (1 + yer) * pv$$

Where:

- **pv** is the present value
- **fv** is the future value

Results were obtained using short real, long real, and temporary real precision numbers when

- **ir** is set to 10% (0.1)
- **np** is set to 365 (for daily compounding)
- **pv** is set to $2,000,000

The results are shown in Table 1.

Table 1. Interest Rate Calculation Results

	yer	Final value
Short real	10.514%	$2,210,287.50
Long real	10.516%	$2,210,311.57
Temp real	10.516%	$2,210,311.57

The times required to calculate these results using FORTRAN 86 with both the 8087 and the 8087 emulator are shown in Table 2.

Table 2. Interest Rate Calculation Times

	8087	Emulator
Short real	1.052 ms	100.4 ms
Long real	1.058 ms	100.7 ms
Temp real	1.041 ms	100.8 ms

The difference in the final value between the short real and long real precision in this simple calculation is $24.07. Although the difference between short and long real precision results shown here is small, this difference would be signficant if the principal was larger, or if the period over which the interest was calculated was longer than a single year. Hence, the long real precision capability of the 8087 can provide most accurate results. Indeed, since the error calculated between the long real precision and temporary real precision results is in the thousandths of cents, the long real results are exactly correct, *to the penny*. Note that temporary real format allows for approximately *18* decimal digits of precision and the full precision of the numbers used in the calculation is not printed in the above table.

Annuities

Values for a frequently used type of annuity were calculated, using routines written in both FORTRAN and Pascal. An annuity is a type of fund which gathers interest at the same time the principal is changing. A mortgage is a type of annuity in which the principal is decreasing, whereas "the sinking fund" implemented here is a type of annuity in which the principal is increasing. In both cases, the interest is added to the principal.

THE SINKING FUND

The "sinking fund" could be characterized by an individual retirement account (IRA). In this fund, a fixed amount is placed in a savings fund each period. This fund also earns a certain amount of interest per period. The problem, then, is to calculate the final value of the

fund (after a certain number of periods). The example given calculates the value after 20 years of a fund in which payments of $1000 are made each month. The annual interest rate is given at 12% (0.12), but the interest is compounded daily.

The first step in solving the problem is to determine the interest rate per month. This is done in a similar manner to the way the effective annual rate is calculated; however, the number of compounding periods is set to the number of days in a month, rather than the number of days in a year. Once this is done, the final value of the annuity is determined by:

$$fv = pmt * \frac{((1 + irp)^{np} - 1)}{irp}$$

Where:

- **fv** is the final value
- **pmt** is the amount placed in the fund each period
- **irp** is the interest rate per period
- **np** is the number of periods

The short, long and temporary real precision results are shown in Table 3.

Table 3. Annuity Calculation Results

	Tot Contrib	Final value	Rate/period
Short	$240,000	$997,103.25	1.005%
Long	$240,000	$997,048.51	1.005%
Temp	$240,000	$997,046.51	1.005%

The times required to calculate these results using FORTRAN 86 with both the 8087 and the 8087 emulator are shown in Table 4. Notice that although the most significant four digits of the interest rates per period shown are the same, the final value using short real precision calculations is inaccurate by $56.74 compared to the final value using long or temporary real calculations.

Table 4. Annuity Calculation Times

	8087	Emulator
Short real	2.121 ms	222 ms
Long real	2.139 ms	229 ms
Temp real	2.106 ms	232 ms

Restocking Algorithms

A restocking algorithm determines when a company should replenish its stock of raw goods which make up its products. A restocking algorithm can be used to determine the restocking pattern if:

- the demand for the given product can be predicted
- carrying costs from month to month are known and fixed
- no shortages are allowed
- lead times are known and fixed

There are three methods commonly used to determine the restocking pattern:

1) the Fixed Economic Order Quantity (EOQ)
2) the Silver-Meal heuristic
3) the Wagner-Whitin method

Of the three, the Wagner-Whitin method is *guaranteed* to provide the *optional restocking pattern,* while the Silver-Meal heuristic may provide a good approximation to this pattern. The fixed Economic Order Quantity will not provide good results when the demand pattern is highly variable. Both the Wagner-Whitin method and the Silver-Meal heuristic are iterative methods in which many options are evaluated before the final restocking pattern is determined.

THE FIXED ECONOMIC ORDER QUANTITY

The simple Economic Order Quantity method may be used to select the number of items to be restocked at a time if the demand is constant. This number is determined by:

$$EQU = \sqrt{\frac{2AD}{vr}}$$

Where:

- A is the set-up cost
- D is the average demand for the period
- v is the variable demand cost per item
- r is the holding cost per item

As this method does not provide for period to period variability in demand, if this demand is variable, the performance of the method will obviously suffer. Its only advantage is simplicity.

THE SILVER-MEAL HEURISTIC

The Silver-Meal heuristic will provide an approximation to the optimal restocking pattern determined by the Wagner-Whitin method. It has been used rather than the Wagner-Whitin in application where better results were required than those supplied by the EOQ method, but where the available computing resources did not allow the use of the Wagner-Whitin method. This

method begins with the first month to be considered, then calculates the total replenishment and holding costs for this month, and a certain number of following months. As the number of months increases, the set-up charge per unit will decrease as it is distributed over more units. Also, however, as the number of units increases, the holding costs will increase. At a certain point, the holding costs will begin to increase at a greater rate than the set-up cost per unit falls. At this point, a "local minimum" of the replenishment cost function will have been realized. The heuristic stops here, and begins the process again with the following month until all the months of the period have been considered. This method may not provide the optimal solution, since it provides only a local minimum, rather than a global minimum. The cost function is not guaranteed to continue to rise once it has begun to rise. This means that the restocking cost may actually fall to a lower level after an initial rise. This method requires much fewer cost calculations than the Wagner-Whitin method, however.

THE WAGNER-WHITIN METHOD

The Wagner-Whitin method is the most computationally intensive method to be discussed. It also is guaranteed to produce the optimal results. It is an application of "dynamic programming." It starts with the last month of the period, determining in inverse order the optimal replenishment pattern for the given month if the inventory is assumed zero at the start of the month. It does this by calculating the replenishment cost for the given month and a number of subsequent months along with the holding costs for the stock replenished in the given month but carried over. The replenishment cost is the sum of the set-up charges and the per unit cost times the number of units acquired. The holding cost is the number of units held but not consumed in a given month. The total stocking costs for this option can then be determined by adding the replenishment cost, the holding cost and the optimal restocking cost for the month following the last one restocked in this iteration (since we have started from the last month of the period, the optimal restocking cost has already been determined for all months following the month being considered). The optimal restocking cost for the last month of the period is the restocking cost for that month alone. For example, if we are trying to determine the optimal restocking pattern from January through December of a year, the determination of the optimal restocking pattern for June might begin like this:

1) Determining restocking cost (startup cost, per part cost, etc.) for June alone.
2) Determine the holding costs (if June alone is being restocked, the holding cost will be zero).
3) Determine the total cost of this option. This will be the restocking cost determined in (1) added to the holding costs determined in (2) added to the optimal restocking cost from zero initial inventory determined previously (using this algorithm) for July.
4) Loop back to (1). However this time, restock for June and July, calculate the holding cost for the July stock, and use the optimal restocking cost from zero initial inventory for August.

This will continue until starting with June, requirements for the balance of the year are being restocked. As the algorithm continues, the cost of each new restocking period (that month and the number of months following it being restocked) for a particular month is compared with a previously determined minimum cost. If it is less, a new minimum cost has been determined, and this restocking pattern will replace the old one as the optimal restocking pattern for the month. As should be apparent, a "horizon" in which the stock will be known to go to zero must be determined in order for this algorithm to be used. While this may at first seem unrealistic, one can see that in any month where the demand for the product is relatively high, the stock will be allowed to go to zero, as the holding cost to that month will surpass the benefit in the restocking cost if the requirements were restocked in the previous month.

OVERALL PERFORMANCE CONSIDERATIONS

Generally, the better an algorithm is in determining an objective function, the greater the computer performance required to execute the algorithm. This is true here, with the most numeric intensive solution guaranteed to realize the optimal solution to the problem, whereas the simpler solutions will only provide approximations to this solution. A more complete explanation of these three methods can be found in Peterson and Silver[2].

EXAMPLE RESTOCKING PROBLEM

Routines were written in Pascal to show possible implementations of the Wagner-Whitin and Silver-Meal heuristic. The EOQ method's results were solved by hand and programmable calculator. The following example was used to demonstrate the results of these methods in solving a general stock management problem:

> A company manufactures video games in which a ROM programmed microcomputer

[2]Peterson, Rein, and Edward A. Silver, *Decision Systems For Inventory Management And Production Planning*, John Wiley & Sons, New York, 1979, pp 308-321.

is used. The manufacturer from which the company buys this microcomputer has an initial ROM set-up charge of $3000, with the cost per part varying from $20 in quantities of less than 500, $17.50 in quantities from 500 to 5000, and $15 in larger quantities. The holding cost is determined to be $0.40 per part. The company barely missed the Christmas rush with its introduction, but has determined that the monthly demand for the next two years will be:

Month	Demand	Month	Demand
January	500	July	3500
February	1500	August	2500
March	2500	September	5000
April	2000	October	7500
May	2000	November	9500
June	1000	December	10000

How should the company restock the microcomputers?

The first problem that must be solved (when using the Wagner-Whitin method) is the horizon to which the stock will be replenished. The criterion to be used is that the final month should be a month in which the demand in the subsequent month is relatively high. Choosing December as the final month would not produce the best results, as the requirements for January are low. Looking at the demand function, it can be seen that the requirements for September are relatively high, so August would be a good choice as the horizon month. It is assumed that the demand for the second year will be similar to the demand predicted for the first year. This allows extending the period of calculation beyond the first year up to the chosen horizon month. Given the total demand function, the part cost, the holding cost, and the startup cost, the problem may be plugged into the Wagner-Whitin, Silver-Meal and Economic Order Quantity methods, and the results calculated.

Using the EOQ with this demand function yields:

- D is 3150
- A is 3000
- v is $15.00
- r is 0.0229

This leads to an EOQ of 7418.

The results obtained from the Wagner-Whitin method, the Silver-Meal heuristic and the EOQ are shown in Table 5. The performance difference between the methods is apparent. Although using the Silver-Meal heuristic would save the business $12,949 over using the EOQ method, using the Wagner-Whitin method would save the business almost $25,000 over using the EOQ (surely below the cost of a small business computer!). The effect on the performance of the Silver-Meal heuristic of choosing a local minimum rather than a global minimum can be seen especially in the first few months in which it replenishes stock 5 times vs. 3 times for the Wagner-Whitin method. It should also be noted that the execution time of the Silver-Meal heuristic using the emulator is still greater than the execution time of the Wagner-Whitin method when the 8087 is used (and the execution time of the EOQ on the hand calculator was much greater than the execution time of either of the two iAPX 86/20 programs!). These results are also interesting when one realizes that until now the Economic Order Quantity method has been the most commonly used method of scheduling stocking intervals.

Linear Programming

Linear programming methods are very powerful ways of finding the optimal solution to operations problems. For example, if a number of different products can be made from a combination of limited resources as expressed by a set of equations, a linear program can be set up to determine the optimal number of each end product to make in order that a certain objective function is maximized. This objective function can be practically anything if it is a linear function—for example, insuring that profit is maximized, that the use of a certain facility is maximized, that shipping costs are minimized, etc. Various software packages are available on the market to solve linear programs. The package which was used in this example consisted of a set of FORTRAN subroutines available from IMSL[3]. To use the routines a FORTRAN program is written to set up the appropriate input arrays and call the routine. They could very easily be integrated into a friendly interactive user environment, where the increased performance of the 8087 would be especially apparent and welcomed.

[3]IMSL, Inc.

Table 5. Restocking Algorithm Results

Wagner-Whitin Method			Silver-Meal Heuristic		Economic Order Quantity	
Month	**Number to Restock**	**Optimal Cost**	**Number to Restock**	**Optimal Cost**	**Number to Restock**	**Optimal Cost**
1	6500	$985,200	500	$996,600	7418	$1,009,549
2	*		1500	$984,850	*	
3	*		7500	$995,600	*	
4	*		*		*	
5	6500	$879,700	*		7418	$888,810
6	*		*			
7	*		6000	$836,500	*	
8	7500	$776,000	*		7418	$769,137
9	*		5000	$742,500	*	
10	7500	$658,500	7500	$664,500	7418	$651,464
11	9500	$543,000	9500	$549,000	14836	$536,525
12	12000	$397,500	19500	$403,500	7418	$308,182
13	*		*		*	
14	*		*		*	
15	7500	$213,100	*		7418	$189,600
16	*		*		*	
17	*		*		*	
18	*		*		*	
19	6000	$94,000	6000	$94,000	3656	$67,980
20	*		*		*	
Total Hold Costs:		$16,200		$19,600		$31,409
Replenishment Costs:		$24,000		$27,000		$24,000
Times Replenished:		8		9		8
Total Cost:		$985,200		$996,600		$1,009,549
Time to calculate above values:						
Using 8087:		310 ms		20 ms		
Using emulator:		22.98 seconds		1.91 seconds		

THE SIMPLEX METHOD

The simplex method is an algorithm which may be used to solve linear programs. The problem is specified to the routine as an objective function (of a certain number of "products") and a set of constraints on the constituents of these products. The objective function specifies exactly how the products are combined to derive the objective function. The constraints specify how each of the constituents are combined to make up each of the products, and also specify the limits imposed on these various constituents.

The set of constraints is usually set up as a two-dimensional matrix, while the objective function is set up as a vector. The combination of the objective function and the set of constraining equations is known as the input tableau. The constraining equations may have both inequality relations (we must use less than 1000 eggs) and equality relations (we must use exactly 1000 eggs). The method itself requires all inequality relations to be converted to equality relations. This is done through the addition of "slack" and "surplus" variables, so called beacuse they fill up the slack or take up the surplus in an inequality relationship. Through many iterations, the method automatically reduces the inequality constraints in the original problem to equality constraints through the addition of these slack and surplus variables. "Artificial" variables are then added to the equation to form an initial set of basic variables or bases. This basis forms a feasible solution to the problem, although this solution is non-optimal. The object, however, is to find the optimal solution to the problem (the solution that optimizes the objective function). This initial form is called the *canonical* form. It transforms the original set of constraint equations and the objective function by the addition of artificial, slack and surplus variables.

After the problem has been set into canonical form, phase I of the problem is ready to begin. In this phase, "pivoting" is performed on the constraint variable matrix until all the coefficients on the modified objective function are less than zero. This pivoting operation is very similar to gaussian elimination. A certain variable in a certain row and column of the matrix is

divided by itself to become 1. Subsequently, every other variable in that row must be divided by this variable. All other variables in the column containing this variable are then eliminated by multiplying the variable set to one by the negative of the variable to be eliminated and then adding the result of this multiplication to the number being eliminated. In order for the matrix to remain valid, this operation must be performed on all other columns of the matrix as well, which leads to a large number of multiplies and divides.

Once phase I is complete, phase II must be initiated. This phase is required if any of the artificial variables remain in the solution as a basis. Through another round of pivoting, the remaining artificial variables are removed from the solution. What finally comes out is the optimal mix of the input variables so the objective function is maximized. A more complete description of both the simplex method and the revised simplex method can be found in Bradley, Hax, and Magnanti[4].

ROUTINE IMPLEMENTED

The linear program used in this example is the IMSL[5] routine "ZX3LP." This routine is the so-called "easy-to-use" linear program solver. It solves the linear program using the revised simplex method. On output, it provides not only the solution to the problem, but also what is called the dual solution. The dual solution gives information about how the solution could be enhanced. The objective function is input to the routine as a vector, while the constraining equations are input to the routine as a matrix. Both inequality and equality constraining equations may be used; the routine will automatically insert slack and surplus variables. The outputs of the routine are two vectors containing the "primal" solution and the dual solution. The routine also calculates the optimal value of the objective function. The version of the routine used was originally developed for the IBM 370/3033 mainframe computer. It required *no* modifications to run on the iAPX 86/20 using FORTRAN 86.

EXAMPLE PROBLEM

The following problem was input to the linear program routine:

> A small cookie company has four different products: chocolate chip cookies without walnuts, chocolate chip cookies with walnuts, brownies without walnuts, and brownies with walnuts. The recipes for the four are:

Chocolate Chip Cookies	Brownies
2 eggs	4 eggs
⅔ cup shortening	⅔ cup shortening
1 cup sugar	2 cups sugar
1 cup brown sugar	
1 tsp. vanilla	1 tsp. vanilla
2¼ cup flour	1¼ cup flour
1 tsp. baking soda	
	1 tsp. baking powder
1 tsp. salt	1 tsp. salt
12 oz. chocolate chips	
	4 oz baking chocolate
(1½ cup walnuts)	(⅔ cup walnuts)
0.15 hour oven time	0.5 hour oven time
0.25 hr mix time (w/o nuts)	0.25 hr mix time (w/o nuts)
0.45 hr mix time (w/nuts)	0.45 hr mix time (w/nuts)

> The available amounts of many of the ingredients have been set previously by contract and may not be altered. They are:

Item	Quantity
eggs	1000
sugar	600 cups
brown sugar	20 cups
baking chocolate	700 oz.
flour	600 subs
baking soda	150 tsp.
baking powder	150 tsp.
chocolate chips	1500 oz.
walnuts	125 cups
oven time	560 hours
mixing time	750 hours

> The amount of profit made for each type cookie is:

Cookie Type	Profit per Batch
chocolate chip w/o	\$0.85
chocolate chip with	\$0.95
brownies w/o	\$1.10
brownies with	\$1.25

> It is assumed that the cookie company can sell everything that it makes. How many of each kind of cookie should the company make in order that the profit is maximized?

The problem was set up into the input tableau. The objective function is:

$$Y = .85*X_1 + .95*X_2 + 1.1*X_3 + 1.25*X_4$$

[4]Stephen P. Bradley, Hax, Arnoldo C., and Magnanti, Thomas L., *Applied Mathematical Programming*, Addison-Wesley, Reading, Massachusetts, 1977.

[5]IMSL, Inc.

Table 6. Example Problem Input Tableau

$2X_1$ +	$2X_2$ +	$4X_3$ +	$4X_4$	=	1000	(eggs)
X_1 +	X_2 +	$2X_3$ +	$2X_4$	=	600	(sugar)
X_1 +	X_2			=	200	(b. sugar)
		$4X_3$ +	$4X_4$	=	700	(b. choc)
$2.25X_1$ +	$2.25X_2$ +	$1.25X_3$ +	$1.25X_4$	=	600	(flour)
X_1 +	X_2			=	150	(b. soda)
		X_3 +	X_4	=	150	(b. powder)
$12X_1$ +	$12X_2$			=	1500	(c. chips)
	$.5X_2$ +	$.65X_4$		=	125	(walnuts)
$.15X_1$ +	$.15X_2$ +	$.5X_3$ +	$.5X_4$	=	560	(oven time)
$.25X_1$ +	$.45X_2$ +	$.25X_3$ +	$.45X_4$	=	750	(mix time)

Where the variable X_1 is the number of batches of chocolate chip cookies without nuts, X_2 is the number of batches of chocolate chip cookies with nuts, X_3 is the number of batches of brownies without nuts, and X_4 is the number of batches of brownies with nuts. The input tableau is shown in Table 6. These were put into the proper input matricies of the ZX3LP program, and the following results were generated:

profit	$299.25
batches of choc chips w/o	70
batches of choc chips with	55
batches of brownies w/o	0
batches of brownies with	150

In addition, the dual solution shows that the single ingredient most limiting the profit of the cookie company is the availability of baking powder, and that for every additional unit (teaspoon) of baking powder available, the profit of the company will increase 1.12 cents.

The calculation times are:

with 8087	1.01 seconds
with emulator	46.78 seconds
with PDP11/45	0.7 seconds
with IBM 3033[6]	0.07 seconds

[6]Non-Intel computers used were a PDP 11/45 mini-computer with 256K MOS RAM, and a FP11-B floating point unit running the UNIX operating system during a period of light load. The program was compiled using the UNIX F77 FORTRAN compiler, and an IBM 370/3033 mainframe computer running the VM/CMS operating system during a period of medium load (the program, however, did not get swapped out of memory during execution). The IBM FORTRAN G compiler was used.

The results show that the performance of the iAPX 86/20 is close to the performance of the mini-computer. In addition, the performance is only a little more than an order of magnitude below the performance of the IBM mainframe, a "maxi" computer with an execution rate of 5 MIPS, and a CPU/hour cost of around $800! A comparison of results between the iAPX 86/20 and the emulator verifies the speed of the 8087 is required to provide results in a reasonable period of time. The power and ease of use of this type of sophisticated numerical method combined with an "electronic worksheet" type of program could be a major advance in the "state of the art" of small business machine software.

CONCLUSIONS

The types of routines demonstrated in this note show that there are many classes of numeric intensive software which are (or should be) commonly used in everyday business operations. With the introduction of the iAPX 86/20, these types of applications are finally within the performance limits of microcomputers selling for a fraction of the cost of the previously required mini- or maxi-computers. In addition, the availability of both Pascal and FORTRAN compilers for the iAPX 86/20 eases the problem of software generation and availability for the processor. Because of the portable nature of these high-level languages, a minimum of effort is required to generate or to port software to the iAPX 86/20 from existing systems. With this kind of numeric intensive software support, the 8087 will be an essential part of the next generation of small business computers.

APPENDIX A

Contents

PAGE

Interest rate calculation routine in FORTRAN A-2

Annuity calculation routine in Pascal A-3

Annuity calculation routine in FORTRAN A-4

Silver-Meal heuristic calculation routine in Pascal A-6

Wagner-Whitin method calculation routine in Pascal A-9

Linear programming routine in FORTRAN A-12

```
FORTRAN-86 COMPILER
:F6:INTST.FOR

SERIES-III FORTRAN-86 COMPILER X023
COMPILER INVOKED BY: FORT86.86   :F6:INTST.FOR

      c
      c       this program provides the yearly effective rate(double and
      c       single precision) and final value when the interest rate
      c       (ir), the number of compounding periods (np),
      c       the present value (pv) are specified.
      c
              real     pv,ir,fv,yer
              real*8   fvd,yerd
              tempreal fvt,yert
              integer*2 np,csv
              integer*4 count,rtimer
      c
      c $2,000,000., at an interest rate of 10% with daily compounding for 1 year
      c
              pv=2000000.
              ir=.1
              np=365
      c
      c set rounding control to single precision
      c
              call stcw87(csv)
              csv=csv .and. #fcffh
              call ldcw87(csv)
      c
              yer=(1+(ir/np))**np - 1
              fv=(1 + yer)*pv
      c
      c set rounding control to double precision
      c
              csv=csv .or. #200h
              call ldcw87(csv)
      c
              yerd=(1+(ir/np))**np - 1
              fvd=(1 + yerd)*pv
      c
      c set rounding control to temp real precision
      c
              csv=csv .or. #100h
              call ldcw87(csv)
      c
              yert=(1+(ir/np))**np - 1
              fvt=(1 + yert)*pv
      c
      c print results
      c
              print *,'single precision: yer=',yer,'fv=',fv
              print *,'double precision: yer=',yerd,'fv=',fvd
              print *,'temp real precision: yer=',yert,'fv=',fvt
              stop
              end
```


AFN-02184A

```
SERIES-III Pascal-86, V1.1


Source File: :F1:ANNP1.PAS
Object File: :F1:ANNP1.OBJ
Controls Specified:  CODE.

             SOURCE TEXT: :F1:ANNP1.PAS
             (*  ANNUITIES: type 1, the sinking fund
              *  if one were to place $1000 a month into a savings fund which
              *  earns 12% per annum, compounded daily, what will be the value
              *  of the fund after 20 years???
              *)
             module annuity;
             public cel;
                 function mqery2x(y,x: real):real;   (* takes y to the x *)
             program annuity(input,output);

             var
                 ir,      (* the annual interest rate *)
                 fv,      (* the final value *)
                 pmt,     (* the amount of the payment *)
                 irp:     (* the interest rate per period *)
                      real;
                 np:      (* the number of periods *)
                      integer;

             begin
             (* insert calculation values *)
                 ir := 0.12;
                 pmt := 1000;

                 np := 12 * 20;       (* 20 years of months *)

             (* calculate the effective interest rate per period *)
                 irp := mqery2x((1+(ir/365.0)),365.0/12.0)-1;
             (* effective monthly rate *)
             (* calculate the future value *)
                 fv := pmt * (mqery2x((1+irp),np)-1)/irp;

             (* print results *)
                 writeln('the effective monthly rate is',irp:18);
                 writeln('the future value of the annuity is',fv:12:2);
                 writeln('the total contribution to the annuity is',np*pmt:12:2);
             end.
```

```
FORTRAN-86 COMPILER
:F1:ANNU1.FOR

SERIES-III FORTRAN-86 COMPILER X023
COMPILER INVOKED BY: FORT86.86  :F1:ANNU1.FOR

c
c  ANNUITIES: type 1, the sinking fund
c       if you place in a savings fund $1000.00 a month, and it
c       earns an interest rate of 12% per annum compounded daily,
c       what will be the value of the fund after 20 years?
c
        real ir,pv,fv,pmt,irp
        real*8 fvd,irpd
        tempreal fvt,irpt
        integer*2 cwv
        integer np

        ir = .12
        pmt = 1000.
c
c the number of periods is the number of months in 20 years!(one period
c       is one month
c
        np = 20*12
c
c set the 8087 to single precision mode
c
        call stcw87(cwv)
        cwv=cwv .and. #fcffh
        call ldcw87(cwv)
c
c first calculate the effective interest rate per period
c
        irp = (1+(ir/365.))**(365./12.) - 1
c
c then calculate the future value
c
        fv = pmt * ((1 +irp)**np - 1)/irp
c
        print *,'single precision values:'
        print *,'the effective rate per month is',irp
        write(6,800)fv
        write(6,801)np*pmt
  800   format('the future value of the annuity is',f18.2)
  801   format('the total contribution to the annuity is',f18.2)
c
c set the 8087 to double precision mode
c
        cwv=cwv .or. #200h
        call ldcw87(cwv)
c
c first calculate the effective interest rate per period
c
        irpd = (1+(ir/365.))**(365d0/12d0) - 1
c
c then calculate the future value
c
        fvd = pmt * ((1 +irpd)**np - 1)/irpd
c
        print *,'double precision values:'
```

```
FORTRAN-86 COMPILER
:F1:ANNU1.FOR

 25           print *,'the effective rate per month is',irpd
 26           write(6,800)fvd
 27           write(6,801)np*pmt
        c
        c set the 8087 to temp real precision mode
        c
 28           cwv=cwv .or. #100h
 29           call ldcw87(cwv)
        c
        c first calculate the effective interest rate per period
        c
 30           irpt = (1+(ir/365.))**(365t0/12t0) - 1
        c
        c then calculate the future value
        c
 31           fvt = pmt * ((1 +irpt)**np - 1)/irpt
        c
 32           print *,'temp real precision values:'
 33           print *,'the effective rate per month is',irpt
 34           write(6,800)fvt
 35           write(6,801)np*pmt
 36           stop
 37           end
```

```
SERIES-III Pascal-86, V1.1


Source File: :F6:SMCT.PAS
Object File: :F6:SMCT.OBJ
Controls Specified: <none>.

          SOURCE TEXT: :F6:SMCT.PAS
          (* This is going to try to find the optimal replacement cost
           *  for a rather variable demand product over 20 months, when
           *  the demand is known, an example could be a video game, using
           *  a single chip ROM programmed microcomputer with an initial set
           *  up charge of $3000.00, demand varies a lot with peak in october
           *  and november(for Christmas), droops in may(vacations), etc.
           *  The cost per part varies from $20.00 per part up to 500,
           *  $17.50 per part from 500 to 5000, and $15.00 above 5,000.
           *  The Sliver-Meal heuristic is going to be used.
           *)
          module silver_meal;
          public timers;
              function rtimer:integer;
              procedure stimer;
          program silver_meal(input,output);
          const
              months = 20;
              monthspl = 21;
              setupcost = 3000.0;
              holdcost = 0.4;
              reallarge = 1.0e10;
              reallargei = 32000;
          var
              repl:           (* first time stock goes to 0 for a given month *)
                  array[1..months] of integer;
              tomake,         (* the number of boxes to make in a month *)
              require:        (* number of boxes required in a given month *)
                  array[1..monthspl] of real;
              trcut,
              holdcostv:      (* holding costs *)
                  array[1..months] of real;
              cost,           (* calculated cost in a given situation *)
              cost1,          (* production cost *)
              cost2,          (* holding cost *)
              totalcost,      (* the total cost of it all *)
              lastcost,       (* used in determining the total cost *)
              totalholdcost:  (* the total hold cost *)
                  real;
              i,j,k:      (* counters *)
                  integer;
              totcnt,         (* accumulated number of boxes in a batch *)
              holdcnt:        (* number of boxed holding *)
                  real;
              count:          (* the 10 ms count *)
                  integer;

          begin
              require[1] := 500;
              require[2] := 1500;
              require[3] := 2500;
              require[4] := 2000;
```

```
SOURCE TEXT: :F6:SMCT.PAS
    require[5] := 2000;
    require[6] := 1000;
    require[7] := 3500;
    require[8] := 2500;
    require[9] := 5000;
    require[10] := 7500;
    require[11] := 9500;
    require[12] := 10000;
    require[13] := 500;
    require[14] := 1500;
    require[15] := 2500;
    require[16] := 2000;
    require[17] := 2000;
    require[18] := 1000;
    require[19] := 3500;
    require[20] := 2500;  (* stop here, because the next month is much
                             higher can assume will restock then *)
    require[monthspl] := reallargei;

    stimer;                           (* start the timer *)

    i := 1;
    while i <= months do begin        (* i is the month working on *)
        trcut[i] := reallarge;
        totcnt := 0;

        j := i;
        while j <= monthspl do begin
            totcnt := totcnt + require[j];
            if totcnt < 500 then costl := 20.0 * totcnt
            else if totcnt < 5000 then costl := 17.5 * totcnt
            else costl := 15.0 * totcnt;
            cost2 := 0.0;
            holdcnt := totcnt;
            for k := i to j - 1 do begin
                holdcnt := holdcnt - require[k];
                cost2 := cost2 + holdcnt * holdcost;
            end;
            cost := (setupcost + cost2 + costl)/(j - i + 1);
            if cost < trcut[i] then begin
                trcut[i] := cost;
                tomake[i] := totcnt;
                holdcostv[i] := cost2;
            end
            else begin
                repl[i] := j;
                i := j;
                j := monthspl;
            end;
            j := j + 1;
        end;
    end;

    count := rtimer;
    j := 1;
```

```
SERIES-III Pascal-86, V1.1


            SOURCE TEXT: :F6:SMCT.PAS
                writeln('month  restock#  optimal cost per period');
                totalcost := 0;
                for i := 1 to months do begin
                    if i = j then begin
                        write(i:5,'   ',tomake[i]:6,'    ',trcut[i]:10:2);
                        writeln(' * restocking now');
                        j := repl[j];
                        lastcost := trcut[i];
                        totalcost := totalcost + lastcost;
                    end
                    else begin
                        totalcost := totalcost + lastcost;
                        writeln(i:5);
                    end;
                end;
                i := 1;
                j := 0;
                totalholdcost := 0.0;
                while i <= months do begin
                    totalholdcost := totalholdcost + holdcostv[i];
                    j := j + 1;
                    i := repl[i];
                end;
                writeln('the total hold cost is',totalholdcost:12:2);
                writeln('stock gets replenished',j:4,' times');
                writeln('replenishment cost is',j*setupcost:12:2);
                writeln('the total cost thingy is',totalcost);
                writeln('the 10 ms count is ',count);
            end.


Summary Information:

PROCEDURE                 OFFSET    CODE SIZE      DATA SIZE      STACK SIZE
SILVER_MEAL                0108H  05F7H  1527D  01ACH    428D  000EH     14D

Total                             06FFH  1791D  01ACH    428D  0042H     66D

 135 Lines Read.
   0 Errors Detected.
 41% Utilization of Memory.
```

```
SERIES-III Pascal-86, V1.1


Source File: :F6:WAGCT.PAS
Object File: :F6:WAGOT.OBJ
Controls Specified: <none>.

            SOURCE TEXT: :F6:WAGCT.PAS
            (* This is going to try to find the optimal replacement cost
             *  for a rather variable demand product over 20 months, when
             *  the demand is known, an example could be a video game, using
             *  a single chip ROM programmed microcomputer with an initial set
             *  up charge of $3000.00, demand varies a lot with peak in october
             *  and november(for Christmas), droops in may(vacations), etc.
             *  The cost per part varies from $20.00 per part up to 500,
             *  $17.50 per part from 500 to 5000, and $15.00 above 5,000.
             *)
            module wag_with;
            public timers;
                function rtimer:integer;
                procedure stimer;
            program wag_with(input,output);
            const
                months = 20;
                monthspl = 21;
                setupcost = 3000.00;     (* mask set up charge *)
                holdcost = 0.4;          (* cost per part of maintaining inventory*)
                reallarge = 1.0e9;
            var
                require,         (* number of chips required in a given month *)
                tomake:          (* the number of chips to make in a month *)
                    array[1..months] of real;
                repl:            (* first time stock goes to 0 for a given month *)
                    array[1..months] of integer;
                optwz:           (* optimum cost for a given month with zero stock
                                         to start with *)
                    array[1..monthspl] of real;
                holdcostv:       (* holding costs *)
                    array[1..months] of real;
                cost,            (* calculated cost in a given situation *)
                cost1,           (* production cost *)
                cost2,           (* holding cost *)
                totalcost,       (* the total cost of it all *)
                totalholdcost:   (* the total hold cost *)
                    real;
                i,j,k:       (* counters *)
                    integer;
                totcnt,          (* accumulated number of chips in a batch *)
                holdcnt:         (* number of boxed holding *)
                    real;
                count:           (* 10 ms count *)
                    integer;

            begin
                optwz[monthspl] := 0;
                require[1] := 500;
                require[2] := 1500;
                require[3] := 2500;
                require[4] := 2000;
```

SERIES-III Pascal-86, V1.1

```
            SOURCE TEXT: :F6:WAGCT.PAS
                require[5] := 2000;
                require[6] := 1000;
                require[7] := 3500;
                require[8] := 2500;
                require[9] := 5000;
                require[10] := 7500;
                require[11] := 9500;
                require[12] := 10000;
                require[13] := 500;
                require[14] := 1500;
                require[15] := 2500;
                require[16] := 2000;
                require[17] := 2000;
                require[18] := 1000;
                require[19] := 3500;
                require[20] := 2500;    (* stop here, because the next month is much
                                         higher can assume will restock then *)

                stimer;
                for i := months downto 1 do begin   (* i is the month working on *)
                    optwz[i] := reallarge;
                    totcnt := 0;
                    for j := i to months do begin   (* j is the option working on *)
                        totcnt := totcnt + require[j];
                        cost1 := setupcost+optwz[j+1];
                        if totcnt <= 500 then cost1 := cost1 + 20.0*totcnt
                        else if totcnt <= 5000 then cost1 := cost1 + 17.5*totcnt
                        else cost1 := cost1 + 15.0*totcnt;
                        cost2 := 0.0;
                        holdcnt := totcnt;
                        for k := i to j - 1 do begin
                            holdcnt := holdcnt - require[k];
                            cost2 := cost2 + holdcnt * holdcost;
                        end;
                        cost := cost1 + cost2;
                        if cost < optwz[i] then begin
                            optwz[i] := cost;
                            repl[i] := j + 1;
                            tomake[i] := totcnt;
                            holdcostv[i] := cost2;
                        end;
                    end;
                end;
                count := rtimer;

                j := 1;
                writeln('month  restock#  optimal cost');
                for i := 1 to months do begin
                    write(i:5,'  ',tomake[i]:6,'    ',optwz[i]:10:2);
                    if i = j then begin
                        writeln(' * restocking now');
                        j := repl[j];
                    end
                    else writeln;
                end;
```

```
SERIES-III Pascal-86, V1.1


          SOURCE TEXT: :F6:WAGCT.PAS
              i := 1;
              j := 0;
              totalholdcost := 0.0;
              while i <= months do begin
                  totalholdcost := totalholdcost + holdcostv[i];
                  j := j + 1;
                  i := repl[i];
              end;
              writeln('the total hold cost is',totalholdcost:12:2);
              writeln('stock gets replenished',j:4,' times');
              writeln('replenishment cost is',j*setupcost:12:2);
              writeln('the 10 ms count is ',count);
          end.


Summary Information:

PROCEDURE                 OFFSET    CODE SIZE      DATA SIZE     STACK SIZE
WAG_WITH                   00E5H  0576H  1398D  01A8H    424D  000EH    14D

Total                             065BH  1627D  01A8H    424D  0042H    66D

 119 Lines Read.
   0 Errors Detected.
 41% Utilization of Memory.
```

```
FORTRAN-86 COMPILER
:F1:COOKIE.FOR

SERIES-III FORTRAN-86 COMPILER X023
COMPILER INVOKED BY: FORT86.86   :F1:COOKIE.FOR

      c
      c this routine will solve a linear problem using the IMSL fortran
      c       library.  the IMSL routine used is "zx3lp" which solves the problem
      c       using the revised simplex method.
      c
              integer ia,n,m1,m2,iw(37),ier
              real*8 a(13,4),b(13),c(4),rw(206),psol(11),dsol(13),s
              integer*4 rtimer,count

             data a/2.,1.,1.,0.,2.25,1.,0.,12.,0.,.15,.25,0.,0.,
           *          2.,1.,1.,0.,2.25,1.,0.,12.,.5,.15,.45,0.,0.,
           *          4.,2.,0.,4.,1.25,0.,1.,0.,0.,.5,.25,0.,0.,
           *          4.,2.,0.,4.,1.25,0.,1.,0.,.65,.5,.45,0.,0./

             data b/1000.,600.,200.,700.,600.,150.,150.,1500.,125.,560.,750.,0.,0./
             data c/.85,.95,1.10,1.25/
      c
      c       n is the number of variables
      c       m1 is the number of inequality constraints
      c       m2 is the number of equality constraints
      c       ia is the declared number of columns of a
      c
              m1 = 11
              m2 = 0
              n = 4
              ia = 13

              print *,'the input tableau:'
              do 100 i=1,ia-2
              write(6,800)a(i,1),a(i,2),a(i,3),a(i,4),b(i)
      800     format(4f10.4,' <= ',f10.4)
      100     continue

              call stimer
              call zx3lp(a,ia,b,c,n,m1,m2,s,psol,dsol,rw,iw,ier)
              count = rtimer()

              print *,'ier = ',ier
              print *,'the final value of the objective function(profit!) is:',s
              print *,'batches of chocolate chip w/o walnuts:',psol(1)
              print *,'batches of chocolate chip with walnuts:',psol(2)
              print *,'batches of brownies without walnuts:',psol(3)
              print *,'batches of brownies with walnuts:',psol(4)
              print *,'the dual solutions follow:'
              do 200 i=1,ia-2
              print *,'var',i,' = ',dsol(i)
      200     continue
              print *,'the calculation time here (in seconds...) is: ',count/100.
              stop
              end
```